OIPE JC67
SEP 24 2004
PATENT & TRADEMARK OFFICE

RECEIVED
OCT 0 1 2004
Technology Center 2600

PATENTS
112025-0180
1611

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In Re The Application of: Andrew McRae	)	
Serial No.: 09/557,480	)	Examiner: Harper, Kevin C.
Filed: April 24, 2000	)	
	)	Art Unit: 2666
For: METHOD FOR HIGH SPEED PACKET CLASSIFICATION	)	

Cesari and McKenna, LLP
88 Black Falcon Avenue
Boston, MA 02210
September 23, 2004

"Express Mail" Mailing-Label Number: EV 335592531 US

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

DECLARATION OF PRIOR INVENTION

TO OVERCOME CITED PUBLICATION UNDER 37 C.F.R. §1.131

To JIM Behmke
No. +1617 951 3927
Company CESARI + McKENNA LLP
From ANDREW McRAE
Company
No. of Pages 4 Date 24th Sept

FAX URGENT Post-it Notes

PURPOSE OF DECLARATION

1. This declaration is to establish completion of the invention of this application in Australia, a WIPO country, at a date prior to August 31, 1999, that is the estimated effective date of the prior art publication, "Packet Classification on Multiple Fields" by Gupta et al., published on or about August 31, 1999, that was cited by the Applicant.

2. The person making this declaration is the inventor.

FACTS AND DOCUMENTARY EVIDENCE

3. To establish the date of completion of the invention of this application, the following attached document is submitted as evidence:

- "TurboACL Software Unit Design Specification," by Andrew McRae, November 18, 1998.

4. From this document, written by the inventor, it can be seen that the invention in this application was made at least by the date of November 18, 1998, which is a date earlier than the effective date of the reference. This document, printed on April 28, 1999, shows the current revision date by the inventor as 11/18/98.

DILIGENCE

5. Applicant acknowledges through this declaration that Applicant acted with diligence in the completion of the invention from the time of his conception, to a time just prior to the date of the reference, up to the filing of this application.

TIME OF PRESENTATION OF THE DECLARATION

This declaration is submitted prior to final rejection.

DECLARATION

6. I, Andrew McRae, as the author of the attached document and the sole inventor of the present invention, hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

SIGNATURE

7.

Full name of sole inventor: Andrew McRae

Inventor's signature: [illegible] Date: 24th Sept 2004

Country of Citizenship: AUSTRALIA

Country of Residence: AUSTRALIA

Post Office Address: 21 GLENCOE CLOSE
BEROWRA NSW 2081
AUSTRALIA

CISCO SYSTEMS

Document Number	ENG-29117
Revision	A
Author	Andrew McRae
Project Manager	Tom Grennan

TurboACL

Software Unit Design Specification

Project Headline

Deterministic lookup of Access Control Lists, for fast packet classification and access checks.

Reviewers

Department	Name
Development Engineering	Reviewer's Name, Reviewer's Title
Development Test Engineering	Reviewer's Name, Reviewer's Title
Customer Advocacy Representative	Reviewer's Name, Reviewer's Title
Compliance Engineering	Reviewer's Name, Reviewer's Title
Impacted Groups	Mail aliases of impacted groups go here (e.g., parser-police)

The departments and/or individuals listed above should be notified in advance and given sufficient time period to review this document. A recommended list of reviewers is located at http://wwwin-eng.cisco.com/Eng/Process/GEM/gem_review_matrix.htm. The requirement for approval is determined by the Project Manager according to the scope of the project.

Modification History

Rev	Date	Originator	Comment
A	11/18/98	Andrew McRae	First Draft

The Specification should be updated as the modules implementing the design are modified.

Definitions

This section defines words, acronyms, and actions which may not be readily understood.

ACL	Access Control List. ACLs are individual filtering rules grouped together in a single list. They are generally used to provide security filtering, though they may be used to also provide a generic packet classification facility.

ACE	Access Control Element. Each individual filtering rule that is part of an ACL is termed an ACE. A group of ACEs form an Access list.
QoS	Quality of Service, where selected packet types are handled differently within the network to provide a differentiated level of reliability, cost etc.
TOS	Type Of Service. A set of flags and values that are part of the IP packet header indicating various parameters related to how the packet should be handled in the network e.g least cost, most reliable path, high priority etc.

1.0 Problem Definition

Access control lists have been in IOS for some time, providing packet classification for a range of applications such as security control, and also QoS related packet matching.

The semantics of access lists are orientated around a list of sequentially searched packet matching rules. Attached to each rule is an action, such as `permit` or `deny`. An access list can be attached to packets received on an interface, and packets outgoing on an interface. One important attribute of access lists is they are searched sequentially, so the rule that is matched is the *first matching rule*, not the rule that provides a best match. Most customers use this attribute in designing their access lists. Implicit in every access list is a final `deny` rule, so that if no rule matches, the default action is used.

IOS access lists may be used with various layer 2 and 3 protocols, but by far the most common are IP access lists. Within IOS, there are two basic types of IP access lists, simple and extended. Simple access lists check only the IP source address of a packet. Extended access lists check a number of header fields, as listed in table 1.

Table 1: IP Extended Access List Fields

Header field	Type of match	Notes
IP Source address	Mask/value	Both source & destination may have non-contiguous masks associated with them.
IP Destination address	Mask/value.	
IP Type of Service	Value - exact match	
IP Precedence	Value - exact match	
IP Protocol	Exact match	Specifies L4 protocol (TCP/UDP/ICMP etc.)
TCP/UDP Source port	Numeric range match	Ports have numeric compares such as equality, inequality, greater than, less than or inclusive range.
TCP/UDP Destination port	Numeric range match	
TCP Flags	Check for RST/ SYN=0	Denoted by `established` keyword.
ICMP type and subtype	Exact match	Allows matching of particular ICMP packets.

Simple and extended ACLs were identified via the access list number; simple access lists ranged from 1 to99, extended ACLs from 100 to 199. Later extensions introduced named access lists, which were extended access lists identified by a name rather than a number.

Many customers employ access lists for security and packet filtering. Access lists are being used that have thousands of rules, and it is not uncommon to have hundreds of rules in an access list. Currently, IOS processes an access list by literally comparing each access list element in turn against a packet until a match is found. The main problem facing IOS in the large scale deployment of access lists is scaling, in terms of performance. Depending on the complexity of each entry, the worst case for matching a packet is when no rule matches, and the implicit deny rule is the default matching rule. This only happens after all rules have been sequentially searched, or O(N) time, where N=number of rules. On a C7200 NPE-200, with approximately 1200 rules, the worst case time to search the ACL is around 760 microseconds. Given that the maximum forwarding rate with no access lists is around 200Kpps, the use of a large access list reduces this performance to around 1300 pps. Even the introduction of smaller access lists causes a significant performance problem. Another issue is the non-determinism of the search, where the processing time of the packet depends entirely upon which rule the packet matches - an early match is found quickly, but a non-match will be done in worst-case time. The non-determinism is worsened by the fact that access list rules can have somewhat arbitrary complexity (by specifying optional fields to be matched).

The problem, then, is how to match a packet against an access list much faster than a sequential search, and also to see if this can be done in a fixed time without regard to the size or complexity of the access list.

2.0 Some Theory

Access lists are now being used for more than just security filtering. They are being used as a way of classifying packets for various route-maps, and quality of service checks. In fact, access lists are a degenerate instance of generic packet classification, which could be described as "a set of rules, with each rule having an associated value, action or class". The action, in the access list instance, is either a permit or deny, but could just as well be a QoS parameter, queuing class, or other unspecified action or value.

So if we examine access lists as simply an instance of the generic packet classification problem, we can start to gain an understanding of the underlying nature of the problem. If we were able to develop a generic solution to the packet classification problem, it would be applicable beyond access lists.

To start with, the action of matching a packet against a database of rules (or ruleset) can be easily considered as an extended memory lookup; if, for instance, the packet header fields that we were interested in total 114 bits (32 bits each IP address, 16 bits each L4 port number, 8 bit protocol, 8 bit TOS/Precedence, 2 bits flags), then if we simply used those 114 bits as an address into a bank of memory, we could be guaranteed that every possible packet would be able to be immediately classified; however, of course, this would require an impossible memory size. And in any case, the full combination of all bits would never occur.

The rule matching semantics themselves do not treat the packet header as a single header, but instead each packet header field is specified separately e.g each rule can optionally specify values for IP source & destination address, IP protocol etc. Some implicit dependance occurs, such as when L4 port numbers are specified, the IP protocol must have been specified as UDP or TCP.

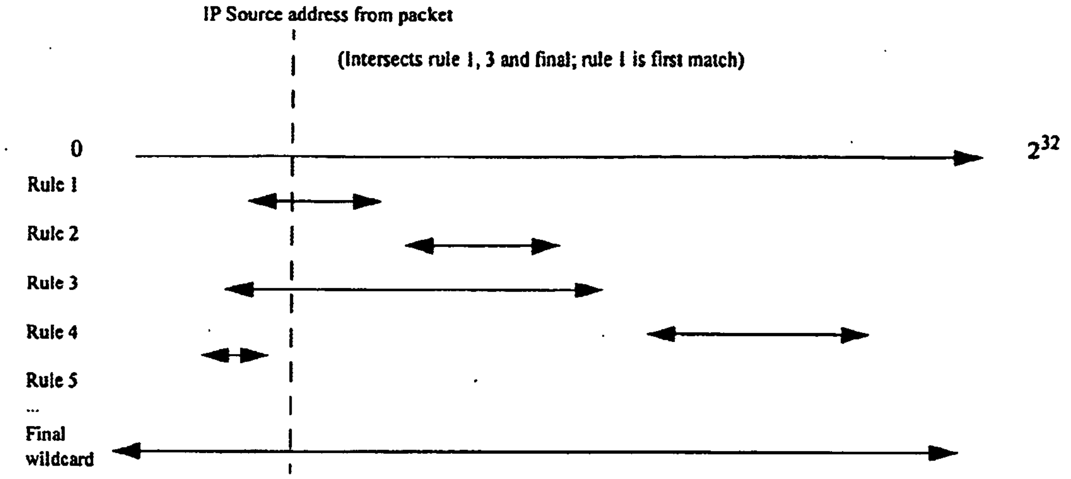
If we just examine one of these fields, such as the 32 bit IP source address, we see that each rule contains either no reference to the field, or an IP address value and associated mask. When no reference to a field occurs in the rule, it is assumed that all values will match (a wildcard). In fact, a wildcard IP address can be treated as simply a value of zero and a mask of all zeroes.

One way of viewing this field is to consider it as an interval, with values along it from 0 to (2^{32}-1), as in Figure 1. Each rule (for the field that we are considering) can be expressed as a subset of that interval, either as a point (a single value with a 255.255.255.255 mask), an interval (a value with a contiguous mask e.g 192.55.99.0/255.255.255.0), or more rarely as a set of intervals (in the case of a non-contiguous mask e.g 192.55.0.1/255.255.0.255).

When a packet is matched against the ruleset, the point on the interval matching the packet field value is used to create an intersecting line. A rule matches this packet field if this line intsersects the interval(s) of the rule. The first rule interval that intersects is the first matching rule.

So the process of determining which rules match can be expressed as the problem of finding which intervals (i.e rules) belong to any particular point on the one-dimensional scale.

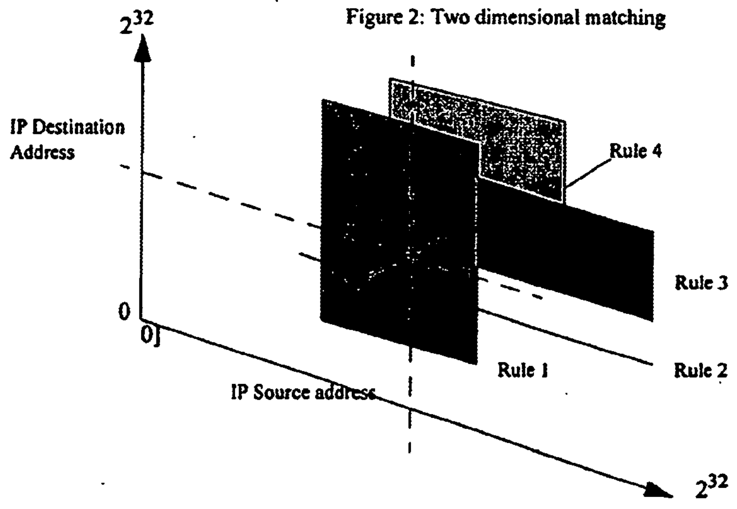
Figure 1: Interval matching


IP Source address from packet
(Intersects rule 1, 3 and final; rule 1 is first match)
0
2^{32}
Rule 1
Rule 2
Rule 3
Rule 4
Rule 5
...
Final wildcard

One attribute of the access lists is that each packet field is disjoint from the others e.g the source and destination IP addresses are specified separately, are matched separately, and have no intrinsic relationship within the packet matching process.

This property can be used to expand our model of matching points in a set of intervals by considering other fields as separate dimensions. Figure 2 shows how a second field (IP destination address) can be added to the existing IP source address dimension.

Figure 2: Two dimensional matching



Each rule has a value/mask for both the IP source and destination address. When plotted on a two dimensional scale, the intervals for each field in the rule sweep out an area that represents the portion of space where the rule matches both fields. Rules that have a 255.255.255.255 mask in both fields are represented by a point, and rules where one field has a 255.255.255.255 mask appear as a line.

The regions that each rule covers indicate the packets that this rule can match.

One observation is that whilst the maximum possible combinations of the two fields is 2^{64}, it can be seen that there are only a few distinct regions where the rules are overlapping, and as a result, for any combination of the two input fields, there will only be a small number of separate regions (i.e sets of rules matching the packet with the two input fields).

The process of expanding the dimensions of the ruleset space can continue from two up to how ever many distinct fields are required in the packet matching e.g IP source & destination, TOS and precedence, protocol, L4 source and destination port, though it would be very hard to diagrammatically show a 7 dimensional space (two is about all my limited drawing skills will allow me). Once the set of regions is discovered, the next step is to select the appropriate region from the matching set.

Matching a packet is thus a case of discovering in which regions a single point lies in an N-dimensional space (given that the rulesets are defining the regions in each dimension, and the packet field values determine the location of the single point in the space).

Given this conclusion, there is considerable literature describing algorithms that address this problem, but most algorithms suffer from either a large memory requirement, or excessive processing. Viewing this problem as a generic mathematical problem, it is indeed very hard to create a deterministic and viable algorithm that deals with the complete range of data; fortunately, the pattern of data both in the rules and in the packets being matched is not random, but is usually limited to a small subset of the possible data range. For example, whilst TCP port numbers can range from 0 to 65535, in reality there are usually only a small number of port numbers of real interest. Often the bulk of port numbers are dealt with in a large range (such as all port numbers greater than 1023).

The result of this subsetting of data is that whilst the possible values of the packet fields may vary greatly, the number of interesting outcomes (or regions that rules represent) is much smaller. This allows a large degree of compression to take place when dealing with the packet fields.

In respect of this background, it can be seen that the sequential searching of the access lists is matching the packet by moving through the different regions of the rulesets, and checking whether the target point (the packet) lies within all the regions of the rule. This is in effect saying, "given a set of regions, check each one to see if this point lies within the region". Turning this around, the far more efficient way of approaching this problem is to say, "given a point (a packet), determine the set of regions (rules) that this point lies in (matches), and then select the best (first) matching region (rule)". This is the approach taken by the TurboACL algorithm, which instead of matching the rules to the packet, uses the packet values to deterministically arrive at the matching rules.

3.0 Design Consideration

Given the nature of packet classification, we have a number of goals that we wish to meet in order to replace the current sequential matching process:

- Any replacement algorithm must preserve the current ACL semantics i.e first match of rules etc.
- The ultimate goal is to have a very fast, deterministic run-time operation which takes a packet and returns the matching rule.
- Depending on the platform, it may be very useful to consider a hardware solution as well. However, for backwards compatibility, any new algorithm must have a viable software only implementation if the algorithm is to be used as a generic IOS improvement.
- The resources required to implement the algorithm in terms of memory or CPU time should not be excessive.
- The scaling properties are very important, so that existing customer ACLs that are very large, or configs that have a large number of ACLs, are usable.
- It would be highly desireable for any changes to be 'drop in' without any change in the customer user interface, nor to the configuration.

The criteria for judging the value of algorithms is:

- The runtime per-packet performance of matching the packet to the rule.
- The memory cost.
- The CPU overhead in maintaining or creating the associated data structures.

- The ability to cater for all existing configs.

ACLs are very pervasive throughout IOS, and any changes done should improve the infrastructure of the packet classification so as to create a reusable service that can be employed by any application.

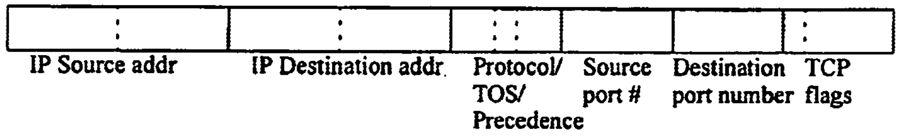
4.0 Algorithm Description

TurboACL has two basic parts to it. The first part is an IOS process that takes the internal access lists in the router's memory and builds a set of data tables. The second part of TurboACL is the run time portion, that is usually part of the packet flow path through the router. This run time operation takes a packet, and uses the packet header values and the data tables calculated by the TurboACL process to lookup which entries in the ACL match this packet. This run time operation is very fast, usually on the order of 0.5 microseconds, and is deterministic i.e no matter how many ACL entries there are, the same number of CPU cycles are consumed.

The hard part is the creation of the data tables, and this is the core of TurboACL. The next section describes the actual mechanics of how these data tables are created.

From our theory section, we see that we can split the packet header fields into separate, disjoint pieces, such as IP source and destination address, protocol, L4 port numbers etc. Whilst the actual boundaries of this split can vary, it is very convenient to split the header in chunks of a consistent size. For a number of reasons, the size of this chunk naturally falls out to be 16 bits - one of the main reasons is that port numbers are 16 bits, and port numbers are dealt with different to IP addresses in that they do not have a value/mask arrangement, but have numeric ranges; thus it is much more convenient to deal with them as a single value rather than splitting them up.

So a packet can be split thus:



Each block is 16 bits. Some smaller header fields are grouped together to form a 16 bit block, and the TCP flags section has only 2 used bits (SYN/RST bits). For protocols such as ICMP, the source port block can be reused as the ICMP type fields.

Each rule in the ACL can be split up into these 8 blocks, and each block dealt with separately.

Given a sample set of ACLs below:

```
access-list 101 deny tcp 192.100.1.0 255.255.255.0 eq smtp
access-list 101 permit ip 192.100.1.0 255.255.255.0
access-list 101 permit ip 192.101.1.0 255.255.255.0
access-list 101 deny tcp any any eq 21
access-list 101 deny udp any any eq 80
```

Taking one of these blocks as an example, say the first part of the IP source address, the ACL entries often contain repeated or wildcard entries, so that you have a set of entries thus:

Table 2: Example ruleset for upper 16 bits of IP source address

Rule number	Value	Mask
1	192.100	255.255
2	192.100	255.255
3	192.101	255.255

Table 2: Example ruleset for upper 16 bits of IP source address

Rule number	Value	Mask
4	0.0	0.0
5	0.0	0.0

The rule is processed by creating a data structure termed an *equivalence set*. Essentially, an equivalence set is the set of unique values that exist across all rules for this particular field. For each entry in the equivalence set, an indication is kept of which rules this entry represents (the idea being that each value may appear in multiple rules e.g the 192.100/255.255 value seen above that appears in both rule 1 and 2) by using a bitmap of which rules are associated with each equivalence set entry. Each equivalence set entry is assigned an identifying index. So for the example above, the following equivalence set is created:

Table 3: Equivalence set

Equivalence set Index	Value/mask	ACL entries (bitmap) 1 2 3 4 5
0	192.100/255.255	1 1 0 0 0
1	192.101/255.255	0 0 1 0 0
2	0.0/0.0	0 0 0 1 1

So we can immediately see that out of the 5 rules, within this field there are only 3 possible outcomes. Going back to our theory, we can see what we are doing is determining how many unique intervals there are in our scale from 0 to $2^{32}-1$, and which rules these are attached to. This preliminary equivalence set reduces the original rules down to a minimal data set, which is then used for the next step.

Our goal is to use the actual packet values to determine which rules match the packet, so the next step is to build a *first level lookup table* that maps the 16 bit packet values to a smaller index value.

The first level lookup table is built by iterating through the 65536 possible values of the packet field; then, for each value, determine which equivalence set entry is applicable, and assign the equivalence set index to this element in the first level lookup table. The equivalence set that was built from the original rules is not directly usable, since there is a 1-to-many relationship between the packet value and the equivalence set i.e a packet value may actually match multiple entries in the equivalence set. For example, a packet value of 0xC064 (192.100) matches equivalence set entries 0 and 2, because entry 2 is a wildcard. So a new equivalence set is created, suitable for use as a target for the first level lookup.

The method for creating this new equivalence set is shown below in psuedo-code:

```
create new equivalence set A
for I = 0..65535 {
      clear bitmap BM
      for each entry in preliminary set {
            if (I matches entry) {
                  BM |= entry.bitmap
            }
      }
      first_level_table[I] = find_match(A, BM)
}
```

The value I is considered to match an entry if that packet header value is covered by that entry's value/mask. For port numbers, instead of value/mask, a numeric range is used. For each value, after the equivalence set entries are scanned, the new bitmap represents all the rules that match this particular packet field value.

The find_match operation will take this bitmap and locate a matching equivalence set entry. If no entry exists with a matching bitmap, a new equivalence set entry is created. Find_match returns the equivalence set index for the new or matching entry.

An example of the new equivalence set is shown below:

Table 4: Packet lookup equivalence set

First level lookup table index values	Equivalence set Index	ACL entries (bitmap) 1 2 3 4 5
0-48995	0	0 0 0 1 1
48996 (192.100)	1	1 1 0 1 1
48997 (192.101)	2	0 0 1 1 1
48998-65535	0	0 0 0 1 1

So for each possible value of the 16 bit field of the packet header being processed, there is an equivalence set index, and the equivalence set entry associated with this index contains a bitmap of which ACL rules match this value.

This is an efficient operation, since the size of the original equivalence set being scanned for matching entries is generally small, and the bitmap processing optimises the creation of new entries. Also, a hash is kept of the bitmap so that ACLs with a large number of rules (and thus a long bitmap) do not cause excessive processing. Note that the new equivalence set has no relationship to the original equivalence set. A variation of this processing could directly scan the rules and set the bitmap bits accordingly, without the creation of a preliminary equivalence set. Which is more efficient depends entirely on the complexity and size of the rule set.

We build an equivalence set for each of the 8 fields for all of the rules. So we have 8 first level lookup tables, with all the entries in the lookup tables referring to the indices of the 8 equivalence sets associated with each packet field.

At this point, the next step is to combine these 8 separate equivalence sets through a technique called *cross-producting* to a set of 4 intermediate lookup tables and equivalence sets, and then to further reduce the resulting tables so that eventually only a single table is left, as in figure 3.

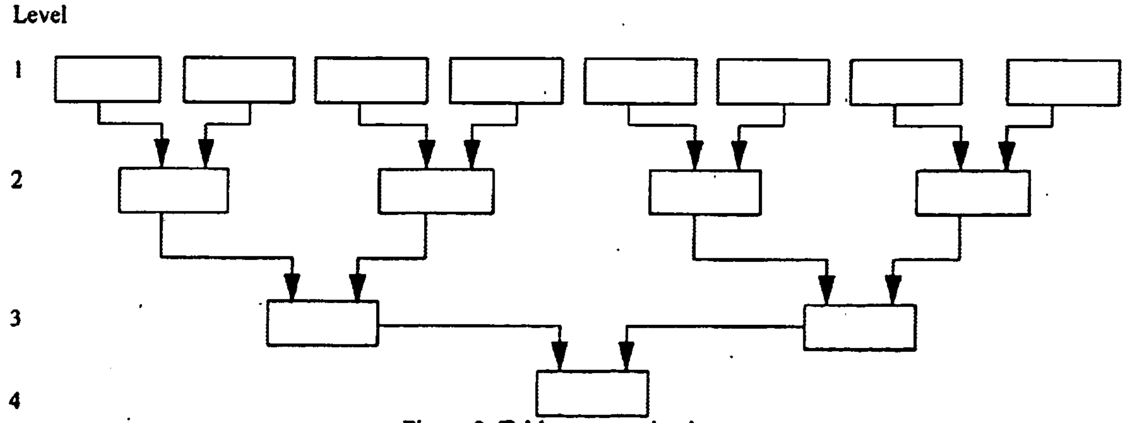


Figure 3: Table crossproducting.

The crossproducting operates by creating a YAFES (yet another equivalence set), iterating through the two sets being combined and creating a cross-product of all the possible outcomes. So, if we have equivalence set A and B we wish to merge, we perform the following processing:

```
create new equivalence set C
for I = all entries in (A) {
        for J = all entries in (B) {
                new bitmap = A[I].bitmap & B[J].bitmap
                lookup-table[I][J] = find_match(C, bitmap)
        }
}
```

The maximum possible size of the new equivalence set is the product of the sizes of the two contributing sets, but in reality the new equivalence set is almost always smaller than the theoretical maximum. The find_match operation is the same as previously described.

Each equivalence set has a lookup table associated with it - this lookup table is used as the cross-product table of the two equivalence sets. For each of the possible combinations of the index values of the two merging equivalence sets, an entry exists in the lookup table containing the new equivalence set's index which is the cross-product of the two contributing entries. The cross-product is calculated by doing an AND on the bitmaps associated with the contributing entries - this is in effect finding the intersection of the two entries, and the bits are set in the bitmap only when *both* contributing entries have the bits set. This discovers the set of ACL rules that match for the particular two values of the contributing equivalence set.

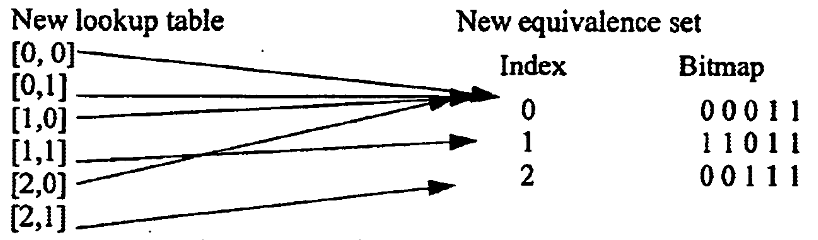
To illustrate this, here are the example equivalence sets from our sample ACL (the second equivalence set is the lower part of the IP source address):

Upper 16 bits of IP source

Index	Bitmap
0	0 0 0 1 1
1	1 1 0 1 1
2	0 0 1 1 1

Lower 16 bits

Index	Bitmap
0	0 0 0 1 1
1	1 1 1 1 1



Index	Bitmap
0	0 0 0 1 1
1	1 1 0 1 1
2	0 0 1 1 1

In essence, what the crossproducting is doing is performing a intersection on the regions covered by the two equivalence sets, and creating a new equivalence set representing all the possible outcomes of the combination of any values in the two packet fields. As an example, say we had a packet with the IP source address of 192.101.1.20, and we used this value as the inputs to our runtime lookup. Using the two 16 bit blocks as indices into the two first level tables for each of these fields, the upper 16 bits would return an equivalence set index of 2, and the lower 16 bit lookup would return a value of 1. To use these results to look up the next level, the two results are used as index values into the the next level's lookup table, obtaining a result of 2. The bitmap associated with this equivalence set entry shows that the rules matching this packet are rules 3, 4 and 5.

As can be seen, though the theoretical maximum size of the new equivalence set is 6, the actual number of possible outcomes is lower (3 in this case) because at each level there is commonality across the rules. If every rule was different from every other in every field, there is no compression, but the nature of the rules (describing real filters for real packets) usually means a high level of compression is achieved.

This process of building the data tables and combining the equivalence sets continues down the levels until a final table is built. This final equivalence set provides all the theoretical possible combinations of rules given any packet header values, and for any of these possible outcomes, there is a bitmap indicating which rules are matching. By doing a find-first-set on the bitmap, the first matching rule can be obtained. Because the bitmaps provide a list of all the rules that match a packet, it is possible to employ this in other ways, such as choosing the best match, or using the list as a set of signatures to check for in intrusion detection. The final lookup can also contain the ACL entry index or a pointer to the actual matching ACL entry to avoid the overhead of searching for the first bit set.

Once the data tables are built, there is no runtime requirement for the bitmaps or for the equivalence set data; only the lookup tables are required. So the equivalence set structures and bitmaps can be freed.

At packet classification time, if all 8 fields are employed, a total of 15 lookups will return the matching ACL entry, regardless of the number of rules in the ACL. If the classification is done in software, these are performed sequentially. It is straightforward, however, to build the lookup tables in hardware. In this situation, the 8 first level lookups can be done in parallel, and the results fed directly to the second level etc. so that a packet can be classified in a time of only 4 memory lookups.

5.0 Algorithm Challenges.

The TurboACL algorithm has been developed and tested on a Unix testbed so that the algorithm can be explored fully before implementation in a router. A number of problems have been discovered and solutions developed. This sections discusses the issues, and outlines the solutions:

1. Exponential cost of creating tables. A small percentage of ACLs are somewhat pathological in nature, in that they cause an exponential rise in the processing time for that ACL. This is usually the result of a large ACL (typically > 500 entries) that has a complex mix of fields; the individual equivalence sets can be large, in the order of hundreds of entries. When two of these equivalence sets are merged, the number of cross-products to be calculated becomes high (possibly tens or hundreds of thousands). Large ACLs also require larger bitmaps, so overall the amount of work that the CPU must perform (to process the large number of cross-products with allocation and comparison of large bitmaps) grows exponentially. Under these circumstances, processing of the ACL data tables may take minutes instead of seconds.
2. High memory overhead. The first level tables that are created by the TurboACL processing map the 16 bit packet header values to the next level's equivalence set indices. If all 8 fields are used, this means there are 8 tables of 65536 entries each for every ACL in the system. If each table entry is 8 bits, that is an overhead of 512Kbytes of memory *per ACL*. If there are a significant number of access lists, the memory overhead is significant. Also, if the number of cross-products is high (similar to the situation described above), the memory requirements for the individual tables can also be high.
3. Run-time calculation of table indices. Whilst the table lookups at runtime can be very low in cost, the merging of the indices can be expensive.

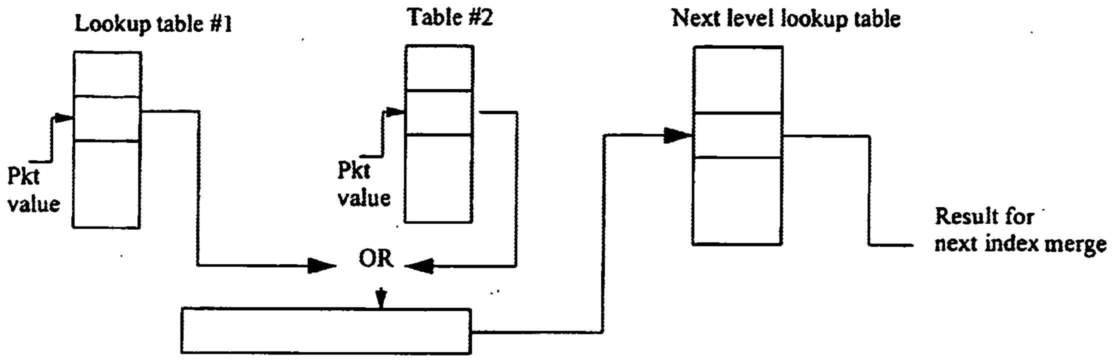
A number of enhancements have been made to the basic TurboACL algorithms to deal with these issues.

The first deals with the detection of ACLs that would be too expensive in CPU performance time or memory usage. This is done by limiting the size of the intermediate lookup tables to a maximum of 65536 entries (16 bits). If a table is detected that would exceed this size, the ACL is not rejected, but the ACL is split in two, and each half of the ACL is processed separately; in effect, two ACLs are processed, and linked so that if the packet is not matched with the first half of the table, instead of an implicit deny being returned, a special entry indicating 'no match' is returned so that the second set of tables can be used. The runtime cost is higher (but not twice as high). There are only a small number of ACLs that need to be split in this way, so it is considered an effective tradeoff.

A side effect of setting a boundary on the size of the lookup tables is that ACLs that would consume excessive CPU to build the data tables are also expensive in memory; splitting the ACL when a problem ACL is recognised also reduces the overall memory requirements for the intermediate tables and the final table.

To simplify the runtime checks required, all intermediate tables are 16 bits wide. This means that all indexes used in the table lookup must be limited to a maximum value of 65535.

Some optimisations exist in the calculation of the first level and intermediate table values so that the runtime overhead of merging separate indices is reduced. The goal is to merge the lookup results from two equivalence sets to form a new index that can be used in the next level down. An initial scheme used a simple 2 dimensional array, but that required a multiply of the major index (but the size of the equivalence set providing the minor index). An alternate scheme rounded the size of the minor index to a power of 2, so that the major index could simply be shifted up a number of bit positions, and the minor index added in. The final scheme that performed best at runtime preshifted the major index lookup table values so that the results of the lookup tables could simply be OR'ed together to form the new index. One downside is that this scheme uses more memory, but the intermediate and final tables are generally not very large, so there is little wastage. So in effect the lookups that occur at runtime are:

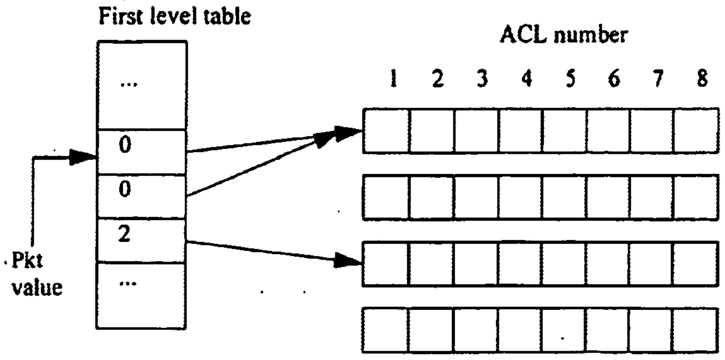


The preshifting of the index values to make them ready for merging does limit their maximum value (due to the fact they are limited to 16 bits). The combined value of the two merging indices must be result in a value 16 bits or less. In reality this is not a problem, as it is been observed that larger tables require too much CPU and memory, so the ACL is split if an intermediate index is calculated to be greater than 16 bits.

Another major challenge has been reducing the memory requirements of the first level tables. With each ACL containing 8 first level tables, and each table containing 65536 entries, the potential memory usage could be very high.

A novel scheme was developed that allows a large number of ACLs to be configured without causing significant memory overhead. It was observed that across the different ACLs, there is a great deal of commonality in the same header fields, but not in different fields e.g across most ACLs, the TCP destination port number is selected from a fairly small number of standard ports, and there are very common patterns (like 'gt 1023'). However, between the TCP destination port and (say) the upper 16 bits of the source IP address, there is no commonality.

Similar to the equivalence set approach, it was seen that if there were 8 separate first level lookup tables, one for each of the header fields, it was possible for the first level table lookup results to point to a set of rows, where each column in the rows equates to a particular ACL:



The contents of the rows are the index values for the next level ready to be merged. The compression comes from the fact that there is typically only a small number of actual rows that are different across ACLs, usually less than 200. This means there can be a single 64K entry table per packet header field, with the contents of the table pointing to the row for this value. There are a couple of interesting implications here:

- 64K entry tables can be allocated for each of the packet fields (8 in total), and as more ACLs are added, the size of the rows that the tables point may grow, and perhaps the number, but the memory growth is far smaller than if 8 new 64K entry tables are added for each ACL.
- Once the initial row lookup is done for each of the 8 packet fields, the values required for each ACL can be picked from the row itself, and *no new first level lookup is required.* So once the row lookup is done, the row pointers can be cached to allow multiple ACLs to be processed without any new packet header access.

These memory compression techniques are enough to resolve any issues of memory requirements. There is a flagfall cost of 512Kb for the 8 first level tables, and thereafter the memory growth requirements are small. A typical large configuration of 21ACLs totalling 3800 entries requires about 1.1Mbytes.

6.0 Interface Design

The TurboACL subsystem is relatively self-contained. There is a process that scans the existing ACLs, and upon finding a candidate for processing, will invoke the TurboACL compilation code to build the data tables for this ACL. To facilitate the runtime selection of how to process an ACL, the access list header will have a function vector added to it. The default entry will process the ACL using the existing code (sequentially searching the ACL). When an ACL has had the TurboACL data structures built and complete, the function vector will be replaced with a TurboACL function, which performs the runtime lookup of the tables, and returns a permit/deny decision.

7.0 End User Interface

There are no changes to the configuration of access lists. It is expected that some new show commands will be added to display internal TurboACL data, and that perhaps a knob will be added to disable TurboACL if required.

8.0 SW Restrictions and Configuration

Initially, it is expected that ACLs containing specialised processing such as reflexive entries will be excluded from TurboACL acceleration. Future enhancements may include them.

9.0 FW Restrictions and Configuration

None known.

10.0 HW Restrictions and Configuration

TurboACL lends itself to hardware implementation, because hardware could take advantage of the inherent parallelism of performing 8 memory lookups at the same time.

11.0 External Restrictions and Configuration

None known.

12.0 Development Unit Testing

Define the approach used in developing the unit test cases.

List the test cases that comprise the unit test plan.

Identify who executes the tests.

Define the unit test completion criteria to meet before feature integration into the development trunk and hand-off to Development Test Engineering.

Identify any special test equipment needs.

References

ENG-11669 - Reflexive ACLs

ENG-16235 - Access Control List Processing in Hardware

ENG-17860 - Cisco TCAM Specification

ENG-21188 - Perseus - ACL Functional Specification

ENG-26071 - Constellation Access Lists

ENG-26939 - ACL Manipulation

ENG-27510 - IP Access Lists for IOS/ENA

ENG-29116 - IOS Access List Infrastructure

Appendixes

Sample code for TurboACL exists in /tftpboot/amcrae/acl.tgz.

Specification Review

It is strongly recommended that this section be used for the specification review minutes, any resulting review action items, the status of those items indicating Close or Open, the dates of closure, etc.

For Template comments please contact the gem-input alias.